An implementation of complete, asynchronous, distributed garbage collection

Fabrice Le Fessant, Ian Piumarta, Marc Shapiro

An implementation of complete, asynchronous, distributed garbage collection

Fabrice Le Fessant, Ian Piumarta, Marc Shapiro

INRIA Roquencourt, B.P. 105, 78153 Le Chesnay Cedex, France

(Email: {*Fabrice.Le_fessant, Ian.Piumarta, Marc.Shapiro*}@inria.fr)

Abstract

Most existing reference-based distributed object systems include some kind of acyclic garbage collection, but fail to provide acceptable collection of cyclic garbage. Those that do provide such GC currently suffer from one or more problems: synchronous operation, the need for expensive global consensus or termination algorithms, susceptibility to communication problems, or an algorithm that does not scale. We present a simple, complete, fault-tolerant, asynchronous extension to the (acyclic) cleanup protocol of the SSP Chains system. This extension is scalable, consumes few resources, and could easily be adapted to work in other reference-based distributed object systems—rendering them usable for very large-scale applications.

Keywords: storage management, garbage collection, reference tracking, distributed object systems.

1 Introduction

Automatic garbage collection is an important feature for modern high-level languages. Although there is a lot of accumulated experience in local garbage collection, distributed programming still lacks effective cyclic garbage collection.

A local garbage collector should be *correct* and *complete*. A distributed garbage collector should also be *asynchronous* (other spaces continue to work during a local garbage collection in one space), *fault-tolerant* (it works even with unreliable communications and space crashes), and *scalable* (since networks are connecting larger numbers of computers over increasing distances).

Previously published distributed garbage collection algorithms fail in one or more of these requirements. In this paper we present a distributed garbage collector for distributed languages that provides all three of these desired properties. Moreover, the algorithm is simple to implement and consumes very few resources.

The algorithm described in this paper was developed as part of a reference-based distributed object system for Objective-CAML (a dialect of ML with object-oriented extensions). Remote references are managed using the Stub-Scion Pair Chains (SSPC) system, extended with our cyclic detection algorithm. Although our system is based on transparent distributed references, our design assumptions are weak enough to support other kinds of distributed languages; those based on channels, for example (π-calculus [8], join-calculus [3], and others).

The next two sections of the paper introduce the basic mechanisms of remote references and the SSPC system for acyclic distributed garbage collection. Section 4 describes our cycle detection algorithm, and includes a short example showing how it works. Section 5 briefly investigates some issues related to our algorithm. Sections 6 and 7 analyze the algorithm in greater depth, and discuss some of the implementation issues surrounding it. The final two sections compare our algorithm with other recent work in distributed garbage collection and present our conclusions.

Conditionally accepted for: 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), 17–19 July, Montreal, Canada.

2 Basics

We consider a distributed system consisting of a set of spaces. Each space is a process, that has its own memory, its own local roots, and its own local garbage collector. A space can communicate with other spaces (on the same computer or a different one) by sending asynchronous messages. These messages may be lost, duplicated or delivered out of order.

Distributed computation is effected by sending messages that invoke procedures in remote objects. These remote procedure calls (RPCs) have the same components as a local procedure call: a distinguished object that is to perform the call, zero or more arguments of arbitrary (including reference) type, and an optional result of arbitrary type. The result is delivered to the caller synchronously; in other words, the caller blocks for the duration of the procedure call. Encoding an argument or result for inclusion in a message is called marshaling; decoding by the message recipient is called unmarshaling.

When an argument or result of an RPC has a reference type (i.e. it refers to an object) then this reference can serve for further RPCs from the recipient of the reference back to the argument/result object. The object is also protected from garbage collection while it remains reachable; i.e. until the last (local or remote) reference to it is deleted.

In the following sections we will write $name_X(A)$ to indicate a variable called *name* located on space X that contains information about object A. We will write a is *increased* to b to mean the variable a is set to the maximum of variable a and variable b.

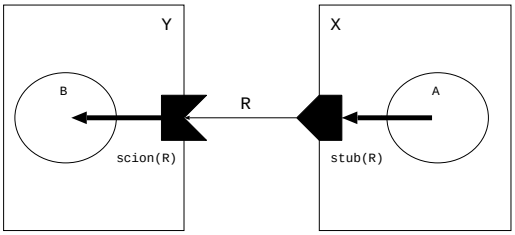

Figure 1: A reference from A in space X to B in space Y.

2.1 Remote references

Marshaled references to local or remote objects are sent in messages to be used in remote computations (e.g. for remote invocation).

Such a reference R from object A in space X to object B in space Y is represented by two objects: $stub_X(R)$ and $scion_Y(R)$. These are represented concretely by:

- a local pointer in X from A to $stub_X(R)$; and
- a local pointer in Y from $scion_Y(R)$ to B.

A scion corresponds to an incoming reference, and is treated as a root during local garbage collection. An object having one or more incoming references from remote spaces is therefore considered live by the local garbage collector, even in the absence of any local reference to that object.

The stub is a local "proxy" for some remote object. It contains the location of its associated matching scion. Each scion has at most one matching stub, and each stub has exactly one matching scion. If several spaces contain stubs referring to some object B, then each will have a unique matching scion in B's space: one scion for each stub.

A reference R is created by space Y and exported to some other space X as follows. First a new scion $scion_Y(R)$ is created and marshaled into a message. The marshaled representation encodes the location of $scion_Y(R)$ relative to X. The message is then sent to to X, where the location is unmarshaled to create $stub_X(R)$.

3 Stub-Scion Pair Chains

The SSPC system [13] is a mechanism for distributed reference tracking similar to Network Objects [1] and supporting acyclic distributed garbage collection. It differs from Network Objects in several important respects, such as: a reduction of the number of messages required for sending a reference, lower latencies, fault-tolerance, and support for object migration. However, we will only describe here the part needed to understand its garbage collector.

The garbage collector is based on **reference listing** (an extension of reference counting that is better suited to unreliable communications), with time-stamps on messages to avoid race conditions.

The following explanation is based on the example shown in Figure 1. This simple example is easily generalizable to situations having more references and spaces.

Each message is stamped by its sender with a monotonically increasing time. When a message containing R is sent by Y to X, the time-stamp of the message is stored in $scion_Y(R)$ in a field called `scionstamp`. When the message is received by X, a field of $stub_X(R)$ called `stubstamp` is *increased to* the time-stamp of the message. For $stub_X(R)$, `stubstamp` contains the time-stamp of the most recent message containing R that was received from Y. Similarly for $scion_Y(R)$, `scionstamp` is the time-stamp of the last message containing R that was sent to X.

When object A becomes unreachable, $stub_X(R)$ is collected by the local garbage collector of space X. When $stub_X(R)$ is finalized, a value called $\texttt{threshold}_X[Y]$ is *increased to* the `stubstamp` field of X. $\texttt{threshold}_X[Y]$ therefore contains the time-stamp of the last message received from Y that contained a reference to an object whose stub has since been reclaimed by the local garbage collector.

After each garbage collection in space X, a message LIVE is sent to all the spaces in the immediate vicinity. The immediate vicinity of space X is the set of spaces that have stubs and scions whose associated scions and stubs are in X. The LIVE message sent to space Y contains the names of all the scions in Y that are still reachable from stubs in X. The value of $\texttt{threshold}_X[Y]$ is also sent in the LIVE message to Y. This value allows space Y to determine the most recent message that had been received by X from Y at the time the LIVE message was sent.

Space Y extracts the list of scion names on receipt of the LIVE message. This list is compared to the list of existing scions in Y whose matching stubs are located in X. Any existing scions that are not mentioned in the list are now known to be unreachable from X, and are called suspect. A suspect scion can be deleted, *provided* there is no danger that a reference to it is currently *in transit* between X and Y.

To prevent an incorrect deletion of a suspect scion, the `scionstamp` field of suspect scions is compared to the $\texttt{threshold}_X[Y]$ contained in the LIVE message. If

$$\texttt{threshold}_X[Y] > \texttt{scionstamp}(scion_Y(R))$$

then some stub referred to by a message sent after the last one containing R has been collected. This implies that the last message containing R was received *before* the LIVE was sent, and so any stub created for R from this message must no longer exist in space X. The suspect scion can therefore be deleted safely.

To prevent out-of-order messages from violating this last condition, any messages from Y marked with a time-stamp smaller than the current value of $\texttt{threshold}_X[Y]$ are refused by space X. ($\texttt{threshold}_X[Y]$ must therefore be initialized with a time-stamp smaller than the time-stamp of the first messages to be received.) This mechanism is called threshold-filtering.

The LIVE message can be extended by a "missing time-stamps" field, to inform the space Y of the time-stamps which are smaller than $\texttt{threshold}_X[Y]$ and which have not been received in a message yet. Y then has the possibility of re-sending the corresponding messages using a new time-stamp and newly-created scions, since older messages will be refused by the threshold-filtering.

The above algorithm does not prevent premature deletion of scions contained in messages that are delayed *in transit*. These deletions are however safe, since such delayed messages will be refused by the threshold-filtering.

This situation can occur only if a more recent message arrives before some other delayed message, *and* the more recent message causes the creation of stubs that are subsequently deleted by a local garbage collection before the arrival of the delayed message. This can not happen with

FIFO communications (such as TCP/IP). Moreover, threshold-filtering of delayed messages is not problematical for applications using unreliable communications (such as UDP), since these applications should be designed to function correctly even in the presence of message loss. Threshold-filtering and message loss due to faulty communication are indistinguishable to the application.

The above distributed garbage collection mechanism is *fault-tolerant*. Unreliable communications can not create dangling pointers, and scions are never deleted in the case of crashed spaces that contain matching stubs (which supports extensions for handling crash recovery). Moreover, it is *scalable* because each space only sends and receives messages in its immediate vicinity, and *asynchronous* because local garbage collections in each space are allowed at any time with no need to synchronize with other spaces.

However, the mechanism is not complete. Distributed cycles will never be deleted because of the use of reference-listing. The remainder of this paper presents our contribution: an algorithm to detect and *cut* distributed cycles, rendering the SSPC garbage collector complete.

4 Detection of free distributed cycles

The detector of free distributed cycles is an extension to the SSPC garbage collector. Spaces may elect to use the acyclic SSPC GC without the detector extension (e.g. for scalability reasons). Spaces that choose to be involved in the detection of cycles are called participating spaces; other spaces are called non-participating spaces. Our detector will only detect cycles that lie entirely within a set of participating spaces.

4.1 Overview

The algorithm is based on date propagation along chains of remote pointers. The useful property of this propagation is that reachable stubs receive increasing dates, whereas unreachable stubs (belonging to a distributed cycle) are eventually marked with constant dates.

A threshold date is computed by a central server. Stubs marked with dates inferior to this threshold are known to have constant dates, and are therefore unreachable. Each participating space sends the minimum local date that it wishes to protect to the central server (stubs with these dates should not be collected). This information is based not only on the dates marked on local stubs, but also on the old dates propagated to outgoing references.

The algorithm is asynchronous (most values are computed conservatively), tolerant to unreliable communications (using old values in computations is always safe, since most transmitted values are monotonically increasing) and benign to the normal SSPC garbage collector (non-participating spaces can work with an overlapping cluster of participating spaces, even if they do not take part in cycle detection).

4.2 Data structures and messages

Stubs are extended with a time-stamp called `stubdate`. This is the time of the most recent trace (possibly on a remote site) during which the stub's chain was found to be rooted. Stubs have a second time-stamp, called `olddate`, which is the value of `stubdate` for the previous trace.

Scions are extended with a time-stamp called `sciondate`. This is a copy of the most recently propagated `stubdate` from the scion's matching stub—i.e. the time of the most recent remote trace during which the scion's chain was found to be rooted. The `stubdate`s from a space are propagated to their matching scions in some other space by sending a STUBDATES message.

STUBDATES messages are stamped with the time of the trace that generated them. Each site has a vector, called `cyclicthreshold`, containing the time-stamp of the last STUBDATES message received from each remote space. The `cyclicthreshold` value for a remote space is periodically propagated back to that space by sending it a THRESHOLD message. The emission of THRESHOLD messages can be delayed by saving the `cyclicthreshold` values for a given time in a set called `CyclicThresholdToSend` until a particular event.

Each site can protect outgoing references from remote garbage collection. For this, it computes a time called `localmin`, which is sent in a LOCALMIN message to a dedicated site, the *Detection Server*, where the minimum `localmin` of all spaces is maintained in a variable called `globalmin`. LOCALMIN messages are acknowledged by the Detection Server by sending back ACK messages.

Finally, to compute `localmin`, each site maintains a per-space value, called `ProtectNow`, containing the new dates to be protected at next local garbage collection. These values are saved in a per-space table, called `Protected Set`, to be re-used and thus protected for some other local garbage collections.

4.3 The algorithm

A Lamport clock is used to simulate global time at each participating space.[1]

4.3.1 Local propagation

The current date of the Lamport clock is incremented before each local garbage collection and used to mark local roots. Each scion's `sciondate` is marked with a date received from its matching stub. These dates are propagated from the local roots and scions to the `stubdate` field of all reachable stubs during the mark phase of garbage collection. If a stub is reachable from different roots marked with different dates then it is marked with the largest date.

Such propagation is easy to implement with minor modifications to a tracing garbage collector. The scions are sorted by *decreasing* `sciondate`, and the object memory traced from each scion in turn. During the trace, the `stubdate` for any visited *unmarked* stub is increased to the `sciondate` of the scion from which the trace began.

4.3.2 Remote propagation

A modified LIVE message, called STUBDATES, is sent to all participating spaces in the vicinity after a local garbage collection. This message serves to propagate the dates from all stubs to their matching scions. These dates will be propagated (locally, from scions to stubs) by the receiving space at next local garbage collection in that space.

[1] A Lamport clock is implemented by sending the current date in all messages. (In our case, only those messages used for the detection of free cycles are concerned). When such a message is received, the current local date is increased to be strictly greater than the date in the message.

```
increment current_date;
FIFO_add(cyclicthresholdtosend_set,
    (current_date,cyclic_threshold[]));

Mark_from_root(local_roots,current_date);
∀ scion ∈ sorted_scions,{
    if scion.scion_date < globalmin then
        scion.pointer := NULL;
    else
        if scion.scion_date = NOW then
            Mark_from_root(scion.pointer,current_date);
        else
            Mark_from_root(scion.pointer,scion.scion_date);
}
∀ space ∈ spaces, {
    ∀ stub ∈ space.stubs, {
        if stub.stub_date > stub.olddate then
            decrease protect_now[space] to stub.olddate;
        stub.olddate := stub.stub_date;}
    FIFO_add(protected_set[space],
        (protect_now[space],current_date));
    protect_now[space] := current_date;
    Send(space,STUBDATES,current_date,
        {∀ stub ∈ space.stubs,
            (stub.stub_id,stub.stub_date)});
}
localmin := min(protected_set[])
Send(server,LOCALMIN,current_date,localmin);
```

Figure 2: Pseudo-code for a local garbage collection. The `Protected Sets` and `Cyclicthreshold-ToSend Set` are implemented by FIFO queues with three functions (add, head and remove).

```
Receive(space,STUBDATES,
    gc_date ,stub_set, threshold) =

increase cyclicthreshold[space] to gc_date;
old_scion_set := space.scions;
space.scions := {};
∀ scion ∈ old_scion_set, {
    find(scion.scion_id,stub_set, found, stub_date);
    if found or scion.scionstamp > threshold then {
        if scion.scionstamp < threshold then
            increase scion.scion_date to stub_date;
        space.scions := space.scions U {scion}
}}
```

Figure 3: Pseudo-code for the STUBDATES handler. The find function looks for a scion identifier in the set of stubs received in the message. If the stub is found in the set then `found` is set to true, and `stub_date` is set to the date on the associated stub. If the `scionstamp` is greater than the `threshold` in the message then the scion is kept alive and its date is not set.

```
Receive(space,LOCALMIN,gc_date,localmin) =

if gc_date>threshold_date[space] then {
    increase threshold_date[space] to current_date;
    localmin[space]:=localmin;
    globalmin:= min(localmin[]);
    Send(space,ACK,gc_date,globalmin);
}
```

Figure 4: Pseudo-code for the Detection Server. The message is treated only if garbage collection date is the lattest date received from the space.

4.3.3 Characterisation of free cycles

Local roots are marked with the current date, which is always increasing. Reachable stubs are therefore marked with increasing dates. On the other hand, the dates on stubs included in unreachable cycles evolve in two different phases. In the first phase, the largest date on the cycle is propagated to every stub in the cycle. In the second phase, no new date can reach the cycle from a local root, and therefore the dates on the stubs in the cycle will remain constant forever.

Since unreachable stubs have constant dates, whereas reachable stubs have increasing dates, it is possible to compute an increasing threshold date called `globalmin`. Reachable stubs and scions are always marked with dates larger than `globalmin`. On the other hand, `globalmin` will eventually become greater than the date of the stubs belonging to a given cycle.

Scions whose dates are smaller than the current `globalmin` are not traced during a local garbage collection. Stubs which were only reachable from these scions will therefore be collected. The normal acyclic SSPC garbage collector will then remove their associated scions, and eventually the entire cycle.

4.3.4 Computation of `globalmin`

`globalmin` is computed by a dedicated space (the *Detection Server*) as the minimum of the `localmin` values sent to it by each participating space.[2] The central server always computes `globalmin` from the most recently received value of `localmin` sent to it from each space. (See the pseudo-code in Figure 4.)

4.3.5 Computation of `localmin`

`localmin` is recomputed after each local garbage collection in a given participating space. (The pseudo-code is shown in Figure 2.)

We now introduce the notion of a *probably-reachable* stub. A stub is probably-reachable either when it has been used by the mutator for a remote operation (such as an invocation) since the last local garbage collection, or when its `stubdate` is increased during the local trace.

This notion is neither a lower nor an upper approximation of reachability. A stub might be both reachable and not probably-reachable at the same time; it might also be probably-reachable and not reachable at some other time. However, on any reachable chain of remote references there is at least one probably-reachable stub for each different date on the chain. Therefore, since each space will "protect" the date of its probably-reachable stubs, all dates on the chain will be "protected".

To detect probably-reachable stubs after the local trace, the previous `stubdate` of each stub (stored the `olddate`

[2] `globalmin` could be computed with a lazy distributed consensus. However, a central server is easier to implement (it can simply be one of the participating spaces), and local networks (where such a collector is most useful) often have a centralized structure.

```
Receive(server,ACK,gc_date,globalmin) =

FIFO_head(cyclicthresholdtosend_set,
    (date,cyclic_thresholds_to_send[]));
if date ≤ gc_date then {
    repeat {
        FIFO_remove(cyclicthresholdtosend_set,
            (date,cyclic_thresholds_to_send[]));
    } until (date == gc_date);
    ∀ space ∈ spaces, {
        Send(space,THRESHOLD,
            cyclic_thresholds_to_send[space]);
};
```

Figure 5: Pseudo-code for the ACK message handler. Old values in the `CyclicthresholdToSend Set` can be discarded, since they are smaller than those which will be sent in the THRESHOLD messages. Their corresponding `ProtectNow` values in the `Protected Sets` will therefore also be removed when the THRESHOLD messages is received.

field), is compared to the newly-propagated `stubdate`. For each participating space in the immediate vicinity, a date (called `ProtectNow`) contains the minimum `olddate` of all stubs which have been detected as probably-reachable since the last local garbage collection.

The value of `ProtectNow` for each space is saved in a per-space set, called `Protected Set`, after each garbage collection. `ProtectNow` is then re-initialized to the current date. The `localmin` for the space is then computed as the minimum of all `ProtectNow` values in all the `Protected Sets`. This new value of `localmin` is sent to the detection server in a LOCALMIN message.

The next value of `globalmin` will be smaller than these `olddates`. All `olddates` associated with stubs that were detected probably-reachable since some of the latest garbage collections will therefore be protected by the new value of `globalmin`: stubs and scions marked with those dates will not be collected.[3]

`globalmin` must protect the `olddates` rather than the `stubdates`. This is because the scions associated with probably-reachable stubs must be protected against collection, and these scions are marked with the `olddate` of their matching stub. In fact `globalmin` not only protects the associated scions, but also all references that are reachable from probably-reachable stubs and which are marked with the `olddates` of these stubs.

4.3.6 Reduction of the Protected Set

STUBDATES and LOCALMIN messages both contain the date of the local garbage collection during which they were sent.

When a STUBDATES message is received (see Figure 3), the per-space threshold `CyclicThreshold` is increased to the GC date contained in the message. The `CyclicThreshold` for each participating space is saved in the `CyclicThresholdToSend Set` before each local garbage collection.

Each LOCALMIN message received by the Detection Server is acknowledged by a ACK message containing the same GC date. When this ACK message is received (see Figure 5) , the `CyclicThresholds` saved in the `CyclicThresholdToSend Set` for the local garbage collection started at the GC date of the ACK message are sent to their associated space in THRESHOLD messages. Older values (for older local garbage collections) in the `CyclicThresholdToSend Set` are discarded (This is perfectly safe. When a space receives a THRESHOLD message it will perform all of the actions that should have been performed for any previous THRESHOLD messages that were lost).

[3]The slightly cryptic phrase "some of the latest garbage collections" will be explained in full in the next section.

```
Receive(space,THRESHOLD,cyclic_threshold) =

FIFO_head(protected_set[space], (protect_now,gc_date));
while (gc_date ≤ cyclic_threshold) {
    FIFO_remove(protected_set[space],
        (protect_now, gc_date));
    FIFO_head(protected_set[space],
        (protect_now, gc_date));
}
```

Figure 6: Pseudo-code for the THRESHOLD handler.

When a `CyclicThreshold` date is received in a THRESHOLD message, all *older* `ProtectNow` values in the `Protected Set` associated with the sending space are removed. (See Figure 6.) These values will no longer participate in the computation of `globalmin`.

We can now explain the cryptic phrase "some of the latest garbage collections" that appeared in the previous section.

The `olddate` on a probably-reachable stub is protected by a `ProtectNow` in a `Protected Set`. It will continue to be protected for a certain time, until several events have occurred. The new `stubdate` must first be sent to the matching scion in a STUBDATES message. From there it is propagated from by a local trace to any outgoing stubs (new probably-reachable stubs in that space will be detected during this trace). The new `localmin` for that must then be received and used by the detection server (ensuring that the `olddates` on the newly detected probably-reachable stubs are protected by next values of `globalmin`). After this, the ACK message received from the detection server will trigger a THRESHOLD message containing a `Cyclicthreshold` equal to the GC date of the STUBDATES message (or greater if other STUBDATES messages have been received before the local garbage collection). Only after this THRESHOLD message is received will the the `ProtectNow` be removed from its `Protected Set`.

4.4 Example

Figures 7, 8 and 9 show a simple example of distributed detection of free cycles.

Spaces A and B are participating spaces; space C is the detection server. The system contains two distributed cycles $C(1)$ and $C(2)$, each containing two objects: $O_A(1)$ and $O_B(1)$ for $C(1)$, $O_A(2)$ and $O_B(2)$ for $C(2)$. $C(1)$ is locally reachable in A, whereas $C(2)$ has been unreachable since date 2. A local garbage collection in A at date 6 has propagated this date to $stub_A(1)$, which was previously marked with date 2. The `Protected Set` associated with B contains a single entry: a `ProtectNow` 2 at date 6.

In figure 7, a local garbage collection occurs in B at date 8. The date 6, marked on $scion_B(1)$, is propagated to $stub_B(1)$ which was previously marked with 2. B saves the

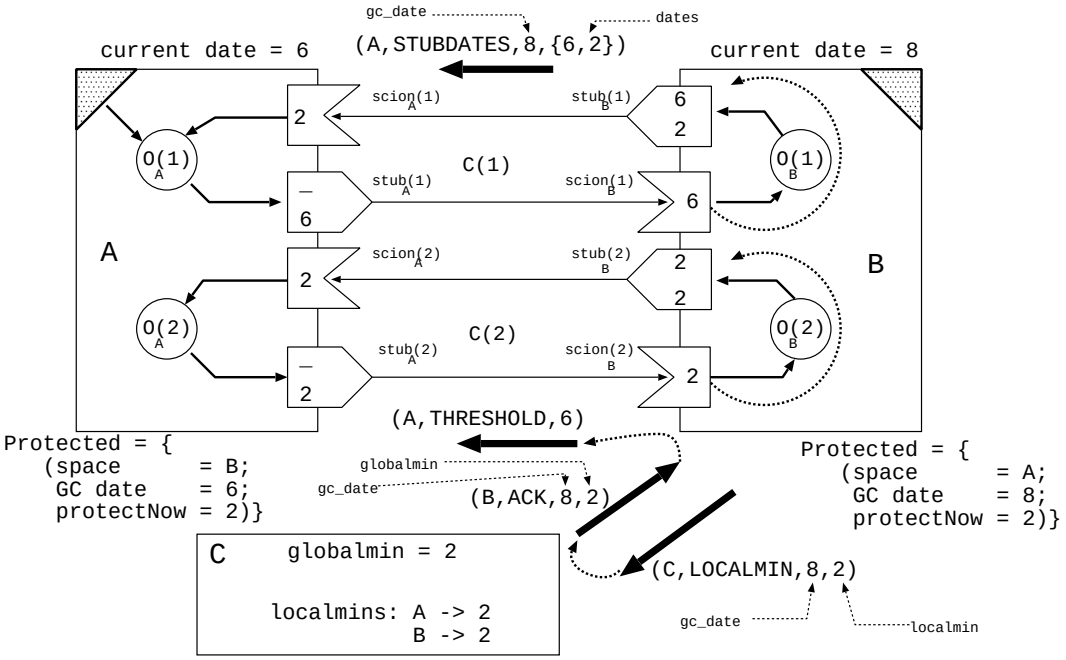

Figure 7: After a local garbage collection at date 8 on space B, the new localmin 2 is sent to the detection server C. After the acknowledgment, the cyclic threshold 6 message is sent to A, which will remove this entry from its protected set.

new `ProtectNow` 2 associated with A in its `Protected Set`. It then sends a STUBDATES message with the new `stubdates` to A, and a `LOCALMIN` message with its new `localmin` 2 to the detection server. After saving this new `localmin`, the detection server sends an ACK message to B containing the same date as the original LOCALMIN message. A `globalmin` value (possibly not up-to-date) can be piggybacked on this message. After reception of this ACK message, B sends a THRESHOLD message to A containing the date of the last STUBDATES message received from A. A consequently removes the associated `ProtectNow` entry from its protected set, which is now empty.

In figure 8, a local garbage collection occurs in A at date 10. The current date 10 is propagated to $stub_A(1)$, previously marked with 6. The `ProtectNow` associated with B is therefore decreased to 6. $stub_A(2)$ does not participate in the computation of `ProtectNow`, since is still marked with 2. This `ProtectNow` is then saved in the `Protected Set`, and the new `localmin` (6) is sent to the detection server. After the reception of the ACK message from C, a THRESHOLD message is sent ot B which removes the associated entry from its `Protected Set`. However, its `localmin` on the detection server is still equal to 2, thus, preventing `globalmin` from increasing.

In figure 9, a local garbage collection occurs in B at date 12. The new `localmin` computed in B is equal to 6. The new `globalmin` is therefore increased to 6. All scions marked with smaller dates will not be traced, starting from the moment that A and B receive this new value of `globalmin`. Consequently $scion_A(2)$ and $scion_B(2)$ will not be traced in subsequent garbage collections, and $O_A(2)$, $O_B(2)$, $stub_B(2)$ and $stub_A(2)$ will be collected by local garbage collections. At the same time, $scion_A(2)$ and $scion_B(2)$ will be collected by the SSPC garbage collector when STUBDATES messages that do not contain $stub_B(2)$ and $stub_A(2)$ are received by A and B respectively. The cycle $C(2)$ has now been entirely collected.

5 Related issues

5.1 New remote references and non-participating spaces

When a new remote reference is created, the stub `olddate` is set to the current date and the `sciondate` is initialized with a special date called `NOW`. Moreover, each time a scion location is resent to its associated space, a new stub may be created if the previous one had already been collected. `sciondate` is therefore re-initialized to `NOW` each time its scion's location is resent in a message.

Scions marked with `NOW` propagate the current date at each garbage collection. A newly-created scion therefore behaves as a normal local root, until a new date is propagated by a STUBDATES message from its matching stub. The SSPC threshold is then compared to the `scionstamp` to ensure that all messages containing the scion have been received before fixing the `sciondate`.

This mechanism is also used to allow incoming references from non-participating spaces. (STUBDATES messages will never be received from non-participating spaces.) The `sciondate`s of their associated scions will therefore remain at `NOW` forever, and they will act as local roots. Distributed cycles that include these remote references will never be collected. This is safe, and does not impact the completeness of the algorithm for participating spaces.

We must also cope with outgoing references to non-participating spaces. We must avoid putting entries in the `Pro-`

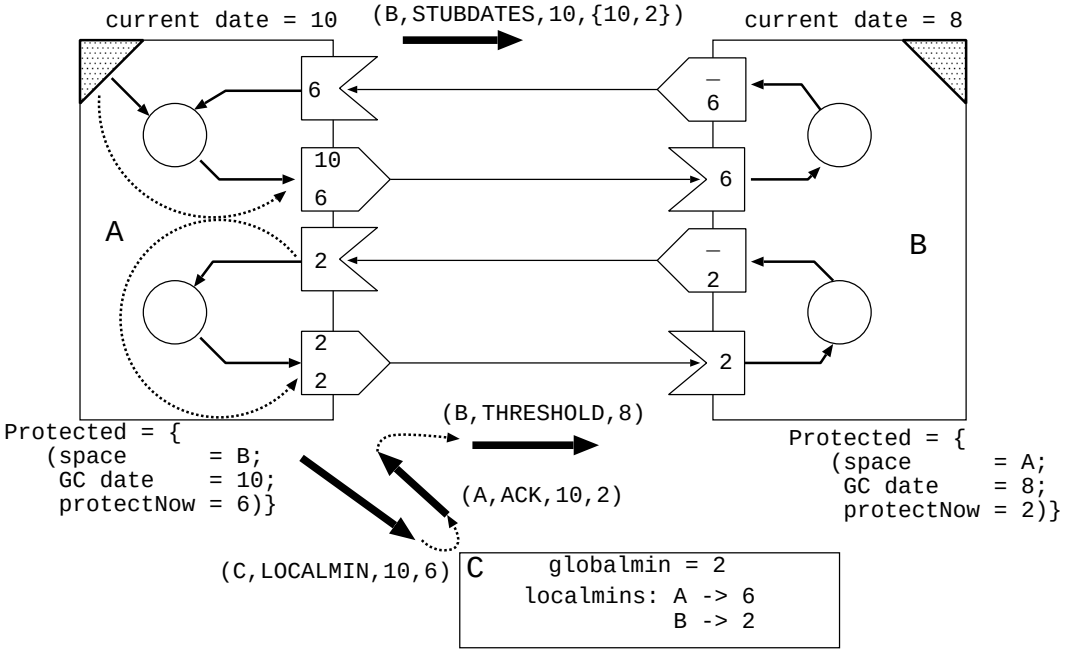

Figure 8: After a new local garbage collection in A, $\texttt{localmin}_A$ is increased to 6.

tected Sets for non-participating spaces, since no THRESHOLD messages will be received to remove such entries. (This would prevent `localmin` and hence `globalmin` from increasing, thus stalling the detection process.) A space must therefore only send STUBDATES messages to, and create entries in the `protected sets` for, known participating spaces. The list of participating spaces is maintained by the detection server, and is sent to other participating spaces whenever necessary (when new participating space arrives, when a space quits the detection process, or if a space is suspected of having crashed or is being too slow to respond).

5.2 Coping with mutator activity

The mutator can create and delete remote references in the interval between local garbage collections. Dates on a remotely-reachable object might therfore never increase because of a "phantom reference": each time a local garbage collection occurs in a space from which the object is reachable, the mutator gives the reference on the object to another space and deletes the local reference — just before the collection. Greater dates might therefore never be propagated to the object and the object would be detected as a free cycle, whereas it is still reachable (see Figure 10 for an example).

Such transient references may move from stubs to scions (for invocation) or from scions to stubs (by reference passing). In the first case, we mark the invoked scions with the current date (This prevents `globalmin` from stalling). In the second case, we ensure that each time a stub is used by the mutator (for invocation, or copy to/from another space) its `olddate` is used to increase the `ProtectNow` associated with the space of its matching scion. The date of the `ProtectNow` therefore always contains the minimum `olddate` of all the stubs that have been used in the interval between two local

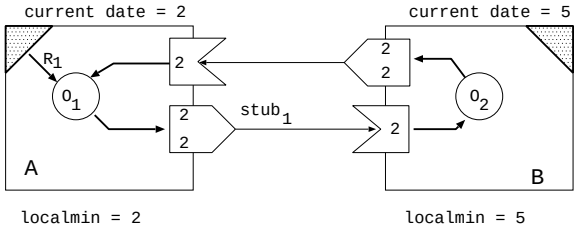

Figure 10: With its local reference to O_1, A invokes $stub_1$ which creates a new local reference in B to O_2. A deletes its local reference R_1, and performs a new local garbage collection. $stub_1$ is therefore re-marked with 2, and $\texttt{localmin}_A$ is increased to 5. This is incorrect, since the cycle is reachable from B. This is the reason why the external mutator activity must be monitored by the detector of free cycles.

garbage collections. This protects any object reachable from these stubs against such transient "phantom references".

5.3 Fault tolerance

Our algorithm is tolerant to message loss and out-oforder delivery. The STUBDATES, THRESHOLD, LOCALMIN and ACK messages are only accepted if their sates are greater than those of the previously received such message. Moreover, the computations are always conservative when using old values. Even LOCALMIN messages may be lost: no ACK messages will be sent and therefore no THRESHOLD will be

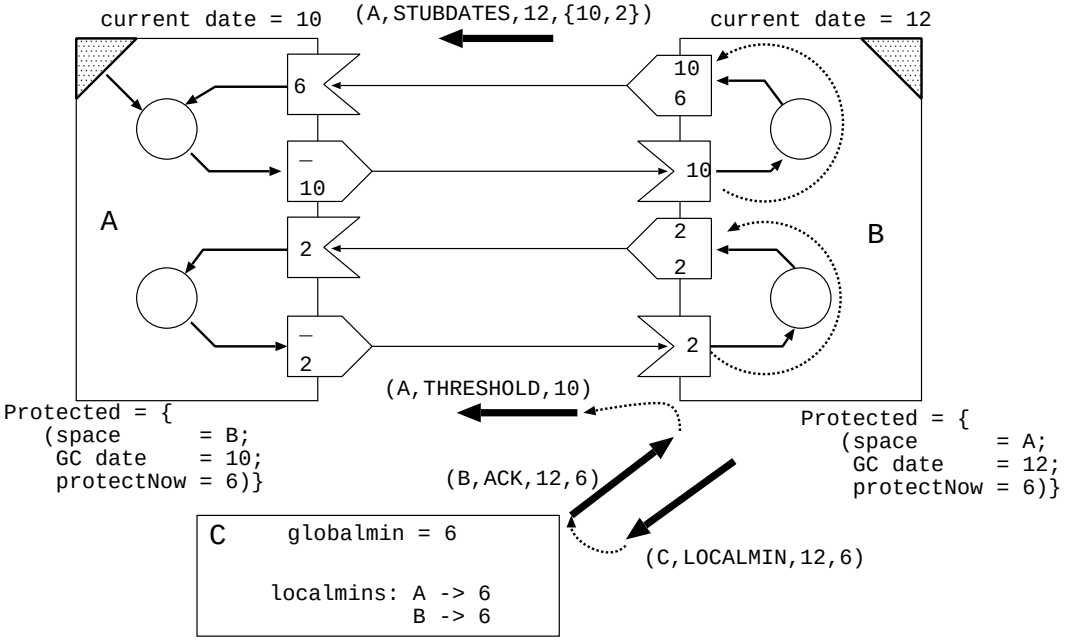

Figure 9: After a new local garbage collection in B, $\texttt{localmin}_B$ is set to 6, and `globalmin` is increased to 6. Thus, the free cycle marked with 2 will be collected since its date is now smaller than `globalmin`.

to other spaces, which will continue to protect the dates that the lost LOCALMIN messages would have protected.

Crashed spaces (or spaces that are too slow to respond) are handled by the detection server, which can exclude any suspected space from the detection process by sending a special message to all participating spaces. The participating spaces set the `sciondates` for scions whose matching stubs are in the suspect space(s) to `NOW`, and remove all entries for the suspected spaces in their `Protected Sets`.

Finally, the detection server may also crash. This does *not* stop acyclic garbage collection, and only *delays* cyclic garbage collection. A detection server can be restarted, and dynamically rebuild the list of participating spaces through some special recovery protocol. It then waits for each participating space to send a new `localmin` value before computing a new up-to-date value for `globalmin`.

6 Analysis

We can estimate the **worst-case time** needed to collect a newly unreachable cycle. It is the time needed to propagate dates greater than those on the cycle to all reachable stubs. Assuming that spaces perform local garbage collections at approximately the same rate, we define a period to be the time necessary for spaces to perform a new local garbage collection. The time needed to collect the cycle is equal to the product of the length of the largest chain of reachable references by the period:

$$time_{collection} = max_{lengths} \times time_{period}$$

We can also estimate the number and the size of the messages that are sent after a local garbage collection. There is one LIVE message (sent by the SSPC garbage collector), plus one STUBDATES message and one THRESHOLD message sent for each space in the immediate vicinity. The first two messages can be concatenated into a single network message. Hence there are only two messages sent for each space in the vicinity. The STUBDATES message contains one identifier and one date for each live stub referring to the destination space, plus the SSPC `threshold` time-stamp. The THRESHOLD message contains only the `CyclicThreshold` value for the destination space.

One LOCALMIN message is also sent to the detection server, and one ACK message sent back from the server.

The `Protected Set` contains triples for each space in the vicinity. For a space X in the vicinity of Y, the number of triples for X in the `Protected Set` of Y is equal to the number of local garbage collections that have occurred on Y since the last garbage collection on X. If the frequencies of the garbage collections in the different participating spaces are similar, the `Protected Set` should not grow too much. If one space requires too many garbage collections, and its `Protected Set` becomes too large, it should avoid performing cyclic detection after each garbage collection (but not stop garbage collections) until sufficient entries in its `Protected Set` have been removed.

Finally, a very large number of spaces may use the same detection server. The server only contains two dates per participating space, and the computation of the minimum of this array should not be expensive.

7 Implementation

Our algorithm has been incorporated into an implementation of the SSP Chains system written in Objective-CAML [5], using the Unix and Thread modules [6].

The Objective-Caml implementation of SSPC consists of 1300 lines of code, of which 200 are associated with the cyclic GC algorithm. The propagation of dates by tracing was implemented as a minor modification to the existing Caml garbage collector [2]. The `Mark_from_root(roots)` function was changed into `Mark_from_root(roots,date)`, which marks stubs reachable from a set of roots with the given date. This function is then applied first to the normal local roots with the current date (which is always greater than all the dates on scions), and then to sets of scions sorted by decreasing dates. Each reachable stub is therefore only marked once, with the date of the first root from which it is reachable.

Finalization of stubs (required for updating the `threshold` when they are collected) is implemented by using a list of pairs. Each pair contains a weak pointer to a stub and a `stubstamp` field. After a garbage collection, the weak pointers are tested to determine if their referent objects are still live. The `stubstamp` field is used to update the `threshold` if the weak pointer is found to be dangling.

The `Protected Set` is implemented as a FIFO queue for each participating space. The head of the queue contains the `ProtectNow` value, which can be modified by the mutator between local garbage collections. When a THRESHOLD message is received, entries are removed from the tail of the queue until the last entry has a date greater than the one in the message. Finally, `localmin` is computed as the minimum of all entries in all queues.

Objective-CAML has high-level capabilities to automatically marshal and unmarshal symbolic messages, easing the implementation of complex protocols. Some modification of the compiler and the standard object library was needed to enable dynamic creation of classes of stubs and dynamic type verification for SSPC. However, these modifications are not related to either the acyclic GC or the cycle detector algorithm.

8 Related work

8.1 Hughes'algorithm

Our algorithm was inspired Hughes' algorithm. In Hughes' algorithm, each local garbage collection provokes a global trace and propagates the starting date of the trace. However, the threshold date is computed by a termination algorithm (due to Rana [11]). The date on a stub therefore represents the starting date of the most recent global trace in which the stub was detected as reachable. If the threshold is the starting date of a terminated global trace, then any stub marked with a strictly smaller date has not been detected as reachable by this terminated global trace. It can therefore be collected safely.

However, the termination algorithm used in this algorithm requires a global clock, instantaneous communication, and does not support failures. Moreover, each local garbage collection in one space triggers new computations in all of the participating spaces. Such behavior is not suitable for a large-scale fault-tolerant system.

8.2 Recent work

Detecting free cycles has been addressed by several researchers. A good survey can be found in [10]. We will only present more recent work below.

All three of the recent algorithms are based on partitioning into groups of spaces or nodes. Cycles are only collected when they are included entirely within a single partition. Heuristics are used to improve the partitioning. These algorithms are complex, and may be difficult to implement. Moreover, their efficiency depends greatly on the choice of heuristic for selecting "suspect objects".

Maheshwari and Liskov's [7] work is based on back-tracing. The group is traced in the opposite direction to references, starting from objects that are suspected to belong to an unreachable cycle. An heuristic based on distance selects "suspected objects". If the backward trace does not encounter a local root, the object is on a free cycle. Their detector is asynchronous, fault-tolerant, and well-adapted to large-scale systems. Nevertheless, back-tracing requires extra data structures for each remote reference. Furthermore, every suspected cycle needs one trace, whereas our algorithm collects all cycles concurrently.

Rodrigues and Jones's [12] cyclic garbage collector was inspired by Lang et al.[4], dividing the network into groups of processes. The algorithm collects cycles of garbage contained entirely within a group. The main improvement is that only suspect objects (according to an heuristics such as Maheshwari and Liskov's distance) are traced. Global synchronization is needed to terminate the detection. It is difficult to know how the algorithm behaves when the group becomes very large.

The DMOS garbage collector [9] has some desirable properties: safety, completeness, non-disruptiveness, incrementality, and scalability. Spaces are divided into a number of disjoint blocks (called "cars"). Cars from different spaces are grouped together into trains. Reachable data is copied from cars in one train to cars in other trains. Unreachable data and cycles contained in one car or one train are left behind and can be collected. Completeness is guaranteed by the order of collections. This algorithm is highly complex and has not been implemented. Moreover, problems relating to fault-tolerance are not addressed by the authors.

9 Conclusion

We have described a complete distributed garbage collector, created by extending an acyclic distributed garbage collector with a detector of distributed garbage cycles. Our garbage collector has some desirable properties: asynchrony between participating spaces, fault-tolerance (messages can be lost, participating spaces and servers can crash), low resource requirements (memory, messages and time), and finally ease of implementation.

It seems well adapted to large-scale distributed systems since it supports non-participating spaces, and consequently clusters of cyclically-collected spaces within larger groups of interoperating spaces.

We are currently working on a new implementation for the Join-Calculus language. Future work includes the handling of overlapping sets of participating spaces, protocols for server recovery, and performance mesurements.

Acknowledgments

The authors would like to thank Neilze Dorta for her study of recent cyclic garbage collectors. We also thank Jean-Jacques Levy and Damien Doligez for their valuable comments and suggestions on improving this paper.

References

[1] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, Asheville, NC (USA), December 1993.

[2] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. of the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, pages 113–123, Charleston SC (USA), January 1993.

[3] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *LNCS*, volume 1119, 1996.

[4] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Proc. of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, Albuquerque, New Mexico (USA), January 1992.

[5] X. Leroy. The objective-caml system software. Technical report, INRIA, 1996.

[6] Xavier Leroy. Unix system programming in caml light. Technical Report No. 147, INRIA, Le Chesnay, France, 1993.

[7] U. Maheshwari and B. Liskov. Collecting distributed garbage cycles by back tracing. In *Principles of Distributed Computing*, 1997.

[8] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes I and II. *Information and Computation*, 100:1 – 40 & 41 – 77, September 1992.

[9] R.L. Hudson R. Morrison J. Eliot B. Moss D.S. Munro. Garbage collecting the world: One car at a time. In *OOPSLA*, Atlanta (U.S.A.), October 1997.

[10] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), September 1995.

[11] S. P. Rana. A distributed solution to the distributed termination problem. *Information Processing Letters*, 17:43–46, July 1983.

[12] Helena Rodrigues and Richard Jones. A cyclic distributed garbage collector for network objects. In *Workshop on Distributed Algorithms (WDAG)*, Bologna (Italy), October 1996.

[13] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, INRIA, Rocquencourt (France), November 1992.